Parallel Programming

15-110 – Friday 03/15

Announcements

- **Hw4** due on Monday
  - Remember – it's extra-large! Don't leave it until the last minute!

- Also remember to complete the midsemester surveys by the Hw4 deadline for bonus points

Learning Goals

- Recognize and define the following keywords: **concurrency**, **parallel programming**, **CPU**, **scheduler**, **throughput**, **multitasking**, **multiprocessing**, and **deadlock**

- Calculate the **total steps** and **time steps** taken by a parallel algorithm

- Create **pipelines** to increase the efficiency of repeated operations by splitting steps across cores

New Unit: Scaling Up Computing

New Unit: Scaling Up Computing

In the unit on Data Structures and Efficiency, we determined that certain algorithms may take a long time to run on large pieces of data.

In this short unit, we'll address the following questions:

- How is it possible for complex algorithms on huge sets of data (like Google search) to run quickly?

- How can we write algorithms that work across multiple computers, instead of running on just one machine?

Two Ways to Increase Efficiency

When we compute Big-O runtimes of programs, we count the number of abstract actions taken. On real computers those actions must be executed using real circuits, and we can influence the speed of those circuits through the design of the computer's hardware. (The same number of actions are taken – they can just happen a lot faster).

There are two ways we can easily speed up a computer:

- Increase the number of **transistors** on the computer

- Have the computer run actions **in parallel** instead of sequentially

Transistors Provide Electronic Switching

A **transistor** is a small device that makes it possible to switch electric signals. In other words, adding a transistor to a circuit gives the computer a **choice** between two different actions.

The more transistors you add to a computer, the faster the computer gets. But you're limited by the **size** of the transistors and the size of the computer; there's only enough space to add so many transistors inside a computer.

Over time, engineers worked to fit more and more transistors on computers. This meant that computers could get faster and faster every year as new advances were made!

Switch From Smaller Transistors to Concurrency

For a while, engineers were able to double the speed of computers every two years by increasing the number of transistors in a computer. However, around 2010 it became physically impossible to keep up this doubling rate because of physical limitations related to power and heat. Computers couldn't get faster unless we made them bigger.

This led to a different tactic: instead of speeding up computers by adding more transistors, we decided to speed up computers by **having them run multiple programs at the same time**. This is called **concurrency**.

To accomplish this, we had to change some of the hardware that makes computers run…

CPUs and Multitasking

Central Processing Units Manage Computation

A **CPU** (aka **core**), is composed of lots of circuits that actually run a program, including:

- **Control unit**: maps the individual steps taken by a program to specific circuits.

- **Logic units**: individual circuits that can perform simple operations (like addition and multiplication).

- **Registers**: store information and act as temporary memory.

Computers also have **memory** that the CPU can read from and write to. This is how the CPU can load instructions and save results.

Combine a CPU with memory and basic mechanisms for input and output, and you've got a simple abstract computer!
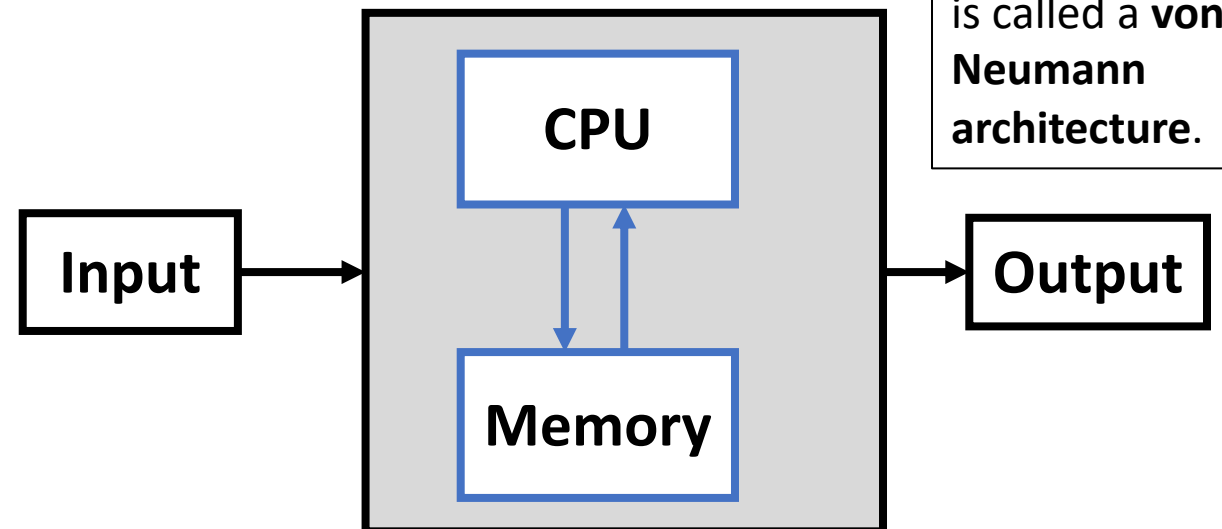

This organization of CPU, memory, input, and output is called a **von Neumann architecture**.

Input → **CPU** ⇅ **Memory** → **Output**

Schedulers Arrange Programs

When you use a computer, you don't just run one program at a time - you likely have multiple applications open and running at any given moment. How does the CPU decide what action to take next?

The **scheduler** is a computer component that takes information from the programs that are currently running and input from the user and decides which program gets to use the CPU.
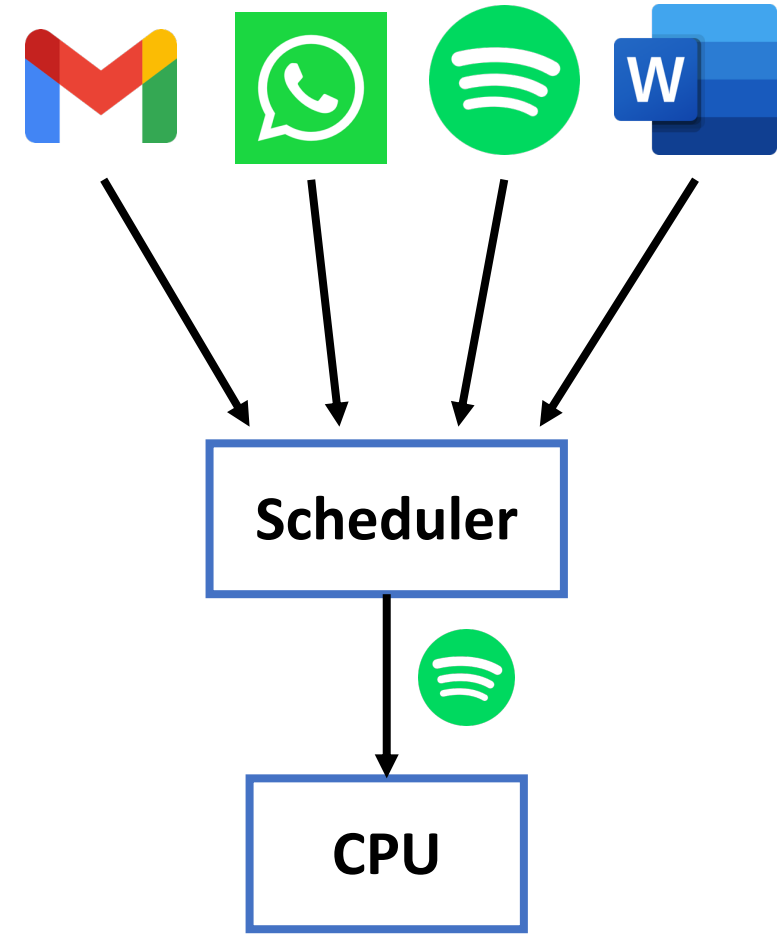
Multitasking with a Scheduler

A scheduler could choose to let a program complete all of its actions before switching to the next program. But it usually doesn't! The scheduler can make programs **appear** to run at the same time by breaking each application's process into steps, then alternating between the steps rapidly.

If this alternation happens quickly enough, it looks like true concurrency to the user, even though only one process is running at any given point in time. This is called **multitasking**.
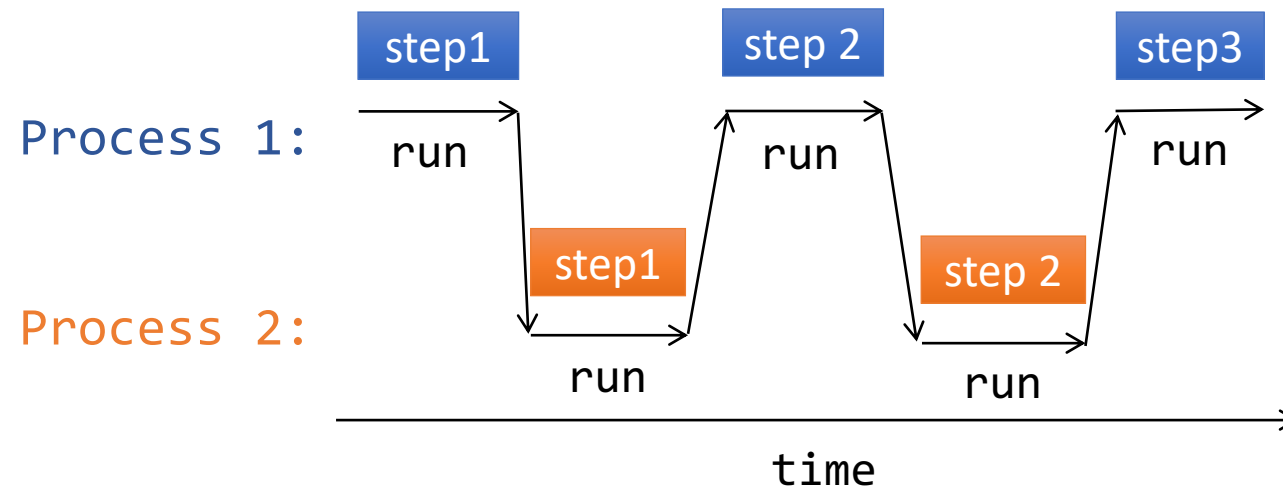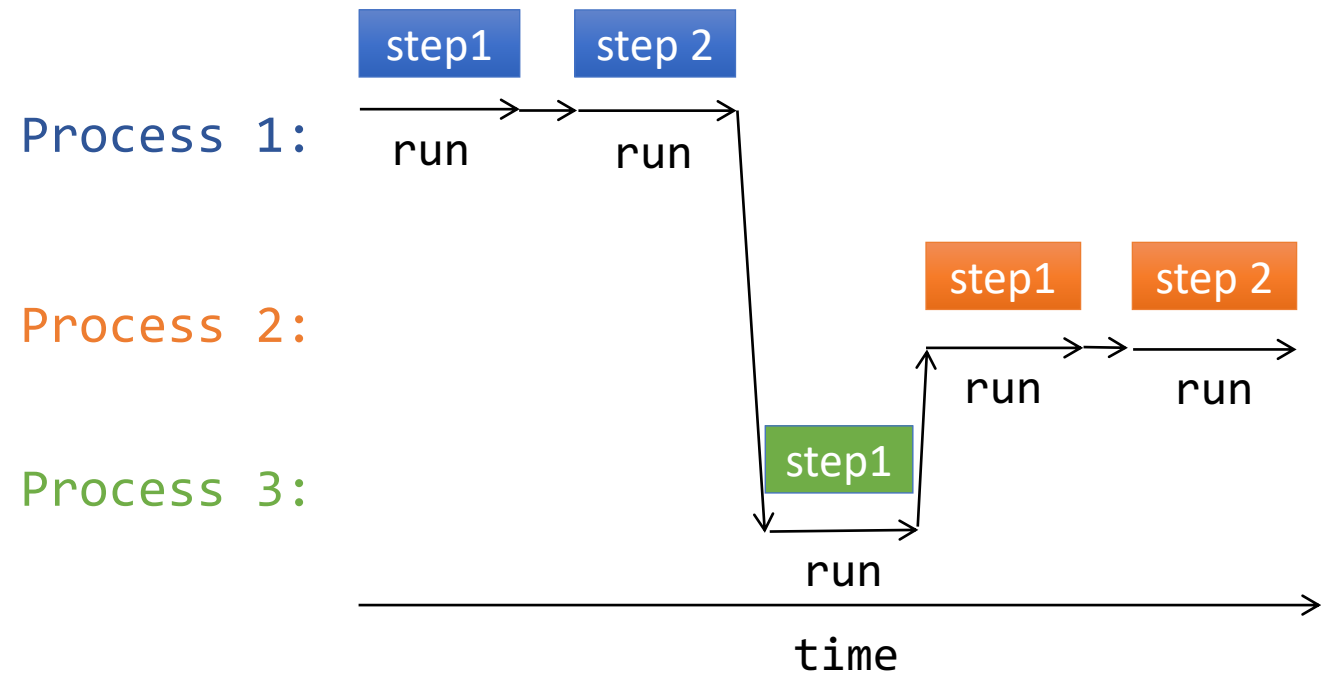
Schedulers Maximize Throughput

When two (or more) processes are running at the same time, the steps don't need to alternate perfectly.

The scheduler may choose to run several steps of one process, then switch to one step of another, then run all the steps of a third. It might even choose to put a process on hold for a long time, if it isn't a priority or is just stalling while waiting for a user action.

In general, the scheduler chooses which order to run the steps in to maximize throughput for the user. **Throughput** is the amount of work a computer can do during a set length of time.

Process 1:

step1   step 2

run   run

Process 2:

step1   step 2

run   run
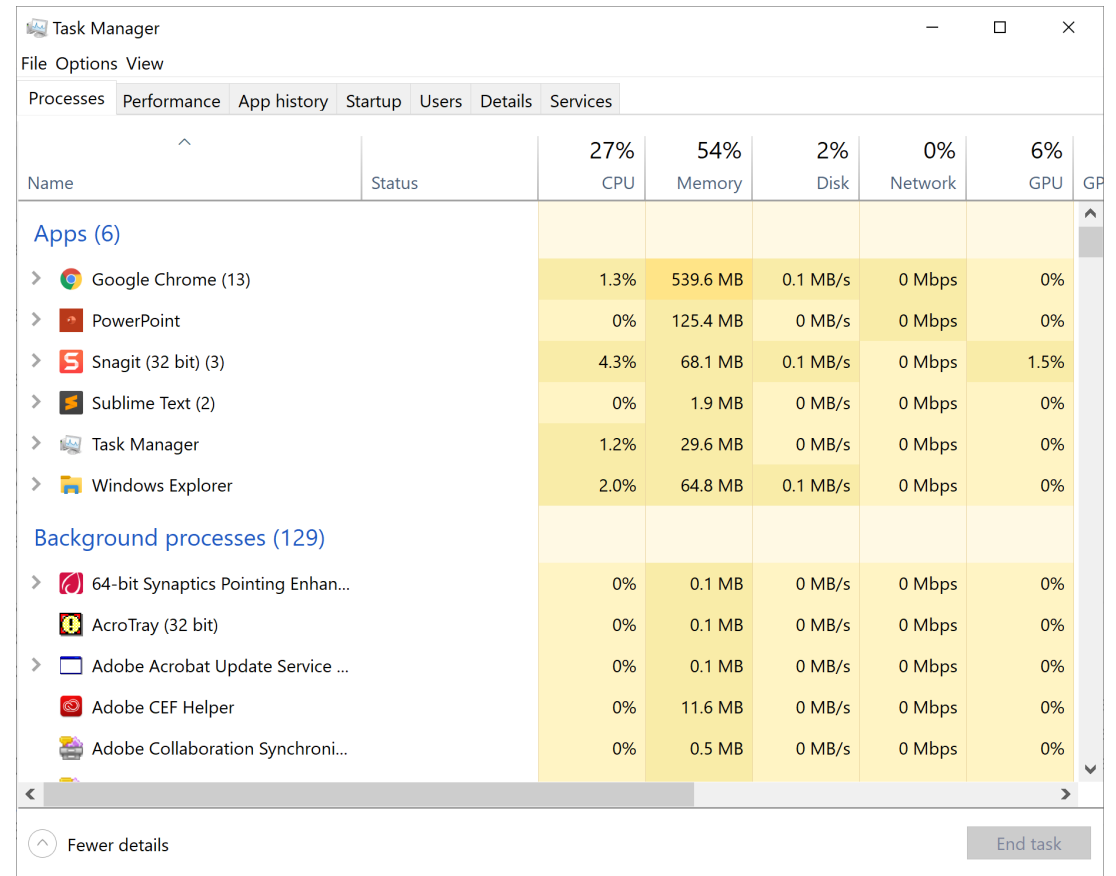
Process 3:

step1

run

time

Your Computer Multitasks

Your computer uses multitasking to manage all of the applications you run, as well as the background processes needed to make your computer work.

You can see all the applications your computer's scheduler is managing by going to your process manager (Task Manager on Windows, Activity Monitor on Mac, htop on Linux). You can even see how much time each process gets on the CPU!

You do: open your process manager now to see how much CPU time each application takes

Multitasking is Fake Concurrency

Multitasking is very useful, but it doesn't increase the speed of a complex algorithm. A single CPU can still only run one action at a time, so an algorithm is still limited to the rate designated by the design of the CPU itself.

How can we speed up algorithms? We can **use multiple CPUs!**

Multiprocessing and Parallel Programming

Multiprocessing Runs Multiple CPUs

Multiprocessing is a method of concurrency where you run multiple actions **at the exact same time** on a single computer.

To make this possible, you put multiple CPUs inside a single computer. Then the computer can send different actions to different CPUs at the same time.
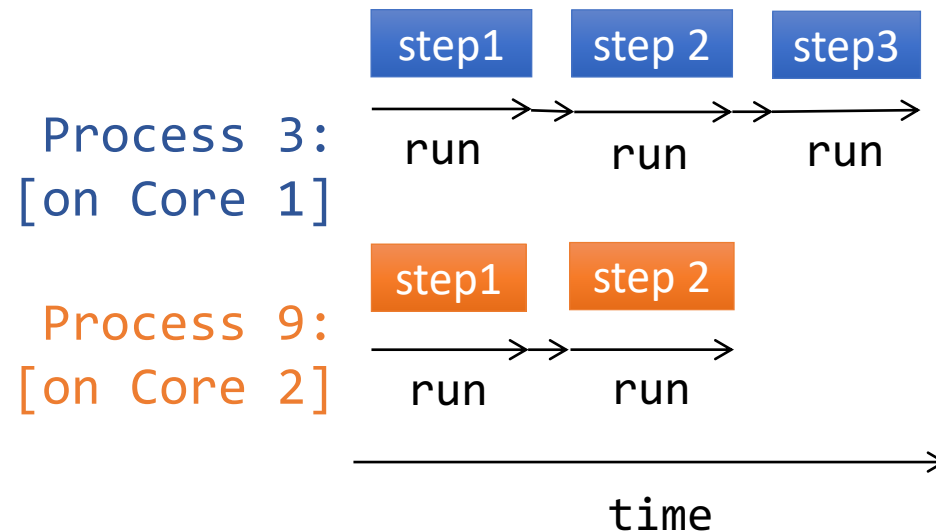
If you have two CPUs instead of one, you can theoretically double the speed of your computer. With four CPUs, you could quadruple it!

Scheduling with Multiprocessing

When we use multiple cores and multiprocessing, we can run our applications simultaneously by assigning them to different cores.

Each core has its own scheduler, so they can work independently.

Multiple cores and multiple processors are slightly different approaches in practice, but we'll treat them as the same in this class.

Process 3:
[on Core 1]

| step1 | step 2 | step3 |

run   run   run

Process 9:
[on Core 2]

| step1 | step 2 |

run   run

time

Simplified Scheduling

Here's a simplified visualization of scheduling with multiprocessing, where we condense all of the steps of an application into one block.

Core 1 | Microsoft Word

Core 2 | Firefox

Core 3 | Thonny

Core 4 | Zoom
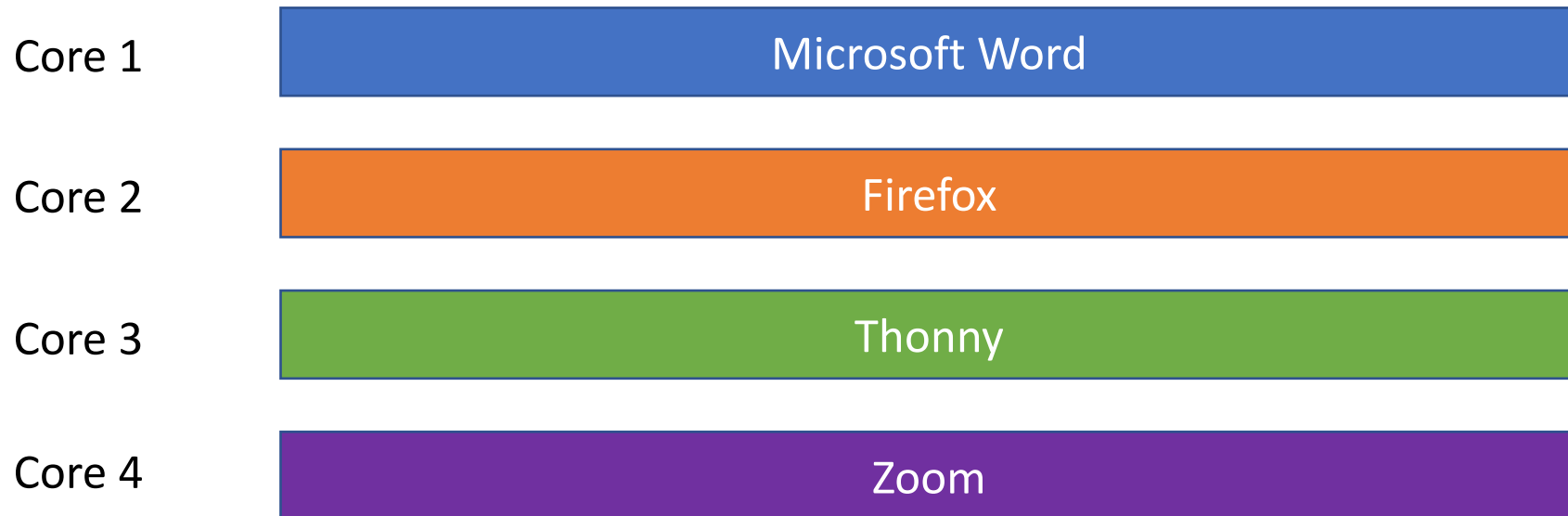
Multiprocessing and Multitasking

The number of cores we have on a single computer is usually still limited. Most modern computers use somewhere between 2-8 cores. If you run more than 2-8 applications at the same time, the cores use **multitasking** to make them appear to run concurrently.

You can check how many cores your own computer has! If you're on Windows, go back to the process manager and switch to the tab 'Performance'. If you're on a Mac, go to About This Mac > System Report > Hardware. On Linux, run `lscpu`.

Scheduling with Multiprocessing and Multitasking

Here's a simplified view of what scheduling might look like when we combine multiprocessing with multitasking.

23
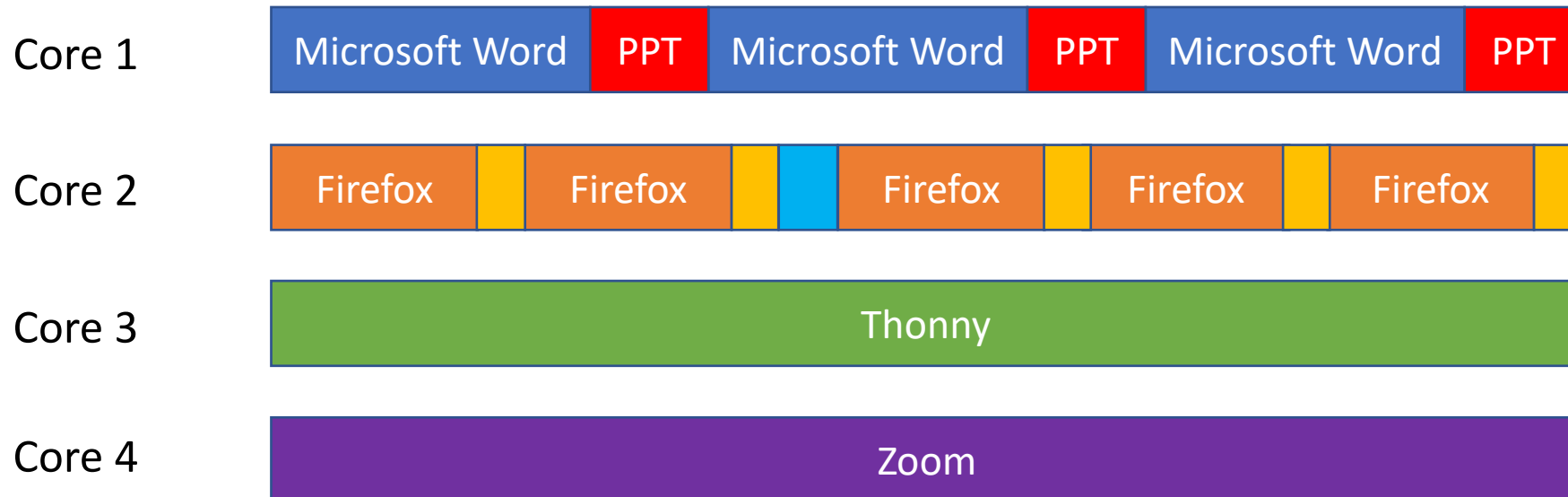
Parallel Programming Divides a Program

Multiprocessing lets us run multiple applications at the same time, but it can also run a single process on multiple cores at the same time. This is called **parallel programming**, because the program is being run on multiple CPUs in parallel.

This should solve our efficiency problem! Now we can take an inefficient algorithm and just split it across a bunch of cores; that way, the work can get done a lot faster!

Unfortunately, it's not that simple...

Difficulty of Design

Parallel programming tends to be more difficult than regular programming. It forces you to think in new ways and adds many new constraints to the problems you try to solve.

Because this is so difficult, we won't write actual parallel programs in this class. But we will talk about common algorithmic approaches for writing parallel code.

To solve a problem using parallel programming, we must design algorithms that can be split across multiple processes. This varies greatly in difficulty based on the problem we're solving!

Summing a Tree Concurrently

Let's start with a simple example. We showed in class how to write a function that can sum all the nodes in a tree. This would run in O(n) time sequentially, since each node needs to be visited. What if we do it concurrently?

We do zero-to-two recursive calls in each recursive case (one on the left child, one on the right). Call the left child recursively on the current core and send the right child's call to a **new** core. This lets us do the two recursive calls **concurrently**. In our example to the right, this is shown using different colors for each core.
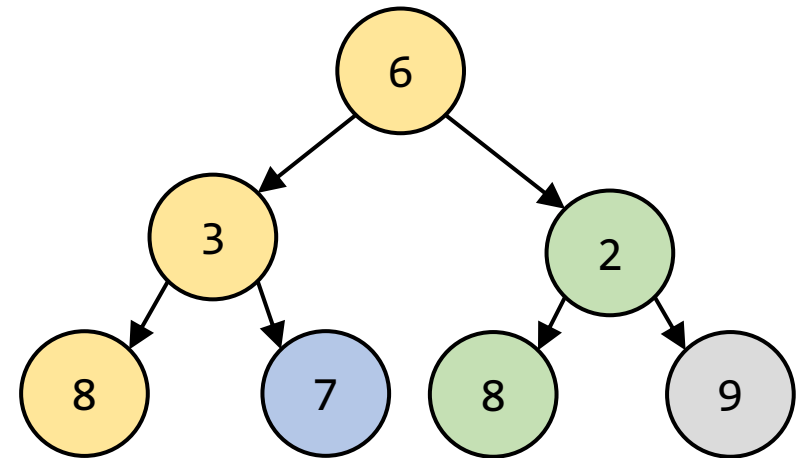
Total Steps vs Time Steps

When we want to determine the **efficiency** of a parallel algorithm, it helps to compare the total number of steps to the number of **time steps**.

The **total** number of steps is just the number of actions taking place across all CPUs in the whole process. For summing the previous tree that's always 7 steps, whether or not we use parallelization.

The number of **time steps** is the number of steps taken **over time**. Multiple actions can be merged into a single time step when they happen at the same time. Summing the tree sequentially takes seven time steps, but summing the tree concurrently only takes **three time steps**.

Summing a Tree Concurrently

How can we calculate the efficiency of concurrent tree summing? Consider the original core, which does the most steps. This will only do one call per level of the tree.

If the tree is balanced, it will have log n levels. **Concurrent tree-summing is O(log n)!**
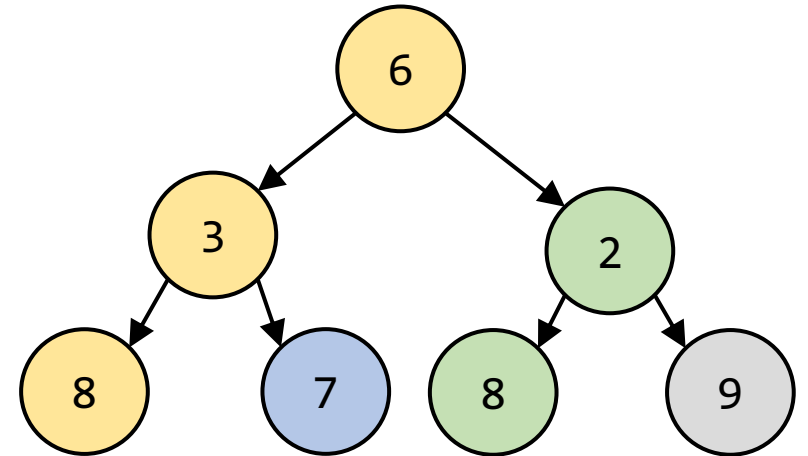
Making Loops Concurrent

It's easy to make recursive problems like tree-summing concurrent if they make multiple recursive calls. It's harder to think concurrently when writing programs that use loops.

We could plan to identify all the iterations of the loop and run each iteration on a separate core. But what if the results of all the iterations need to be combined? And what if each iteration depends on the result of the previous one? This gets even harder if we don't know how many iterations there will be overall, like when we use a while loop.

A bit later, we'll talk about how to use algorithmic plans to address these difficulties.

```
def search(lst, target):
    for item in lst:
        if item == target:
            return True
    return False
```

```
def getSum(lst):
    sum = 0
    for item in lst:
        sum = sum + item
    return sum
```

```
def powersOf2(n):
    i = 2
    while i < n:
        print(i)
        i = i * 2
```

Sharing Resources

The next difficulty of writing parallel programs comes from the fact that multiple cores need to **share individual resources** on a single machine.

For example, two different programs might want to access the same part of the computer's memory at the same time. They might both want to update the computer's screen or play audio over the computer's speaker.
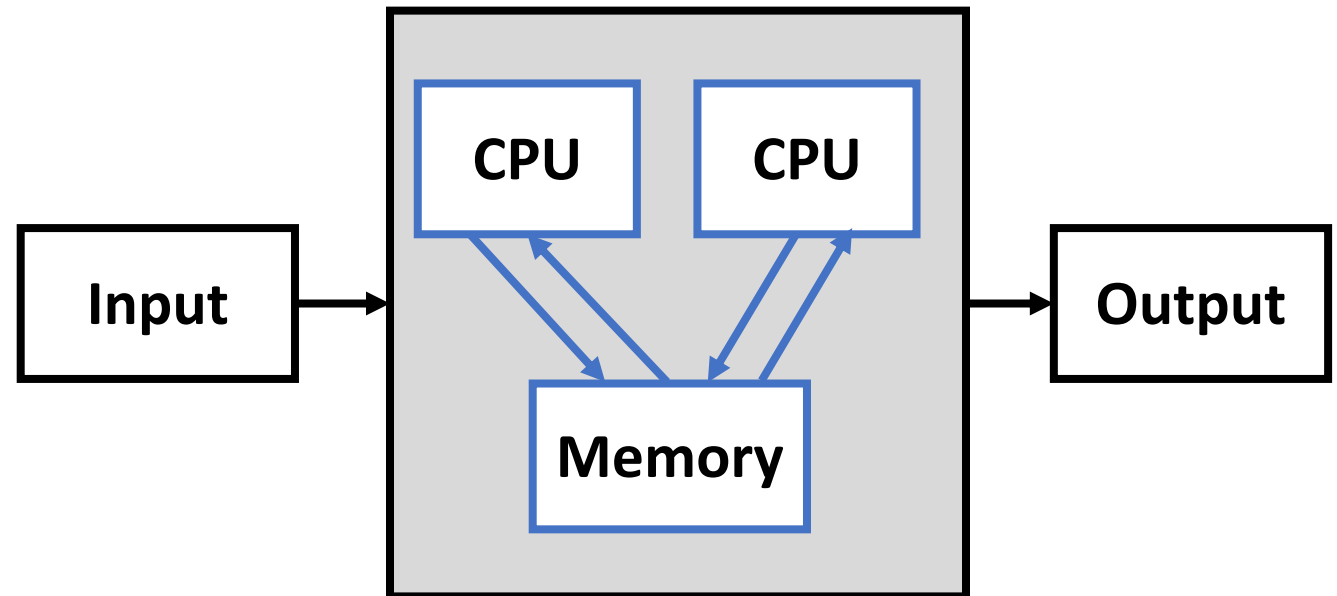
Locking and Yielding Resources

We can't just let two programs update a resource simultaneously- this will result in garbled results that the user can't understand. For example, if one program wants to print "Hello World" to the console and the other wants to print "Good Morning", the user might end up seeing "Hello Good World Morning".

To avoid this situation, programs put a **lock** on a shared resource when they access it. While a resource is locked, no other program can access it.

Then, when a program is done with a resource, it **yields** that resource back to the computer system, where it can be sent to the next program that wants it.

Sidebar: if we want two programs to use a resource simultaneously, we usually use a third program to combine the actions together, and that third program is the one that accesses the resource. For example, if you listen to music while watching a lecture recording, your computer **mixes** the two audio tracks together and plays the combined result.

Deadlock Stalls the System

In general, this system of locking and yielding fixes most cases where programs might try to use a resource at the same time. But there are some situations where it can cause trouble.

Two programs, Youtube and Zoom, both want to access the screen and audio. They put their requests in at the same time, and the computer gives the screen to Youtube and the audio to Zoom.

Both programs will lock the resource they have, then wait for the next resource to become available. Since they're waiting on each other, they'll wait forever! This is known as **deadlock**.
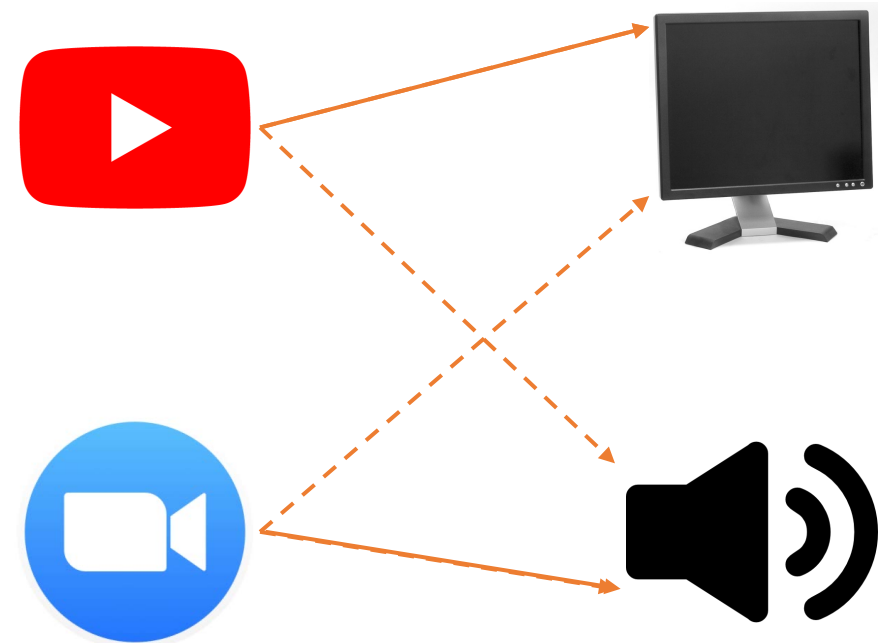
Deadlock Definition

In general, we say that deadlock occurs when two or more processes are all waiting for some resource that other processes in the group already hold. This will cause all processes to wait forever without proceeding.

Deadlock can happen in real life! For example, if enough cars edge into traffic at four-way intersections, the intersections can get locked such that no one can move forward.

In the example to the right, each direction of traffic needs two of the intersection spots, but only has one. All four directions are blocked as a result.
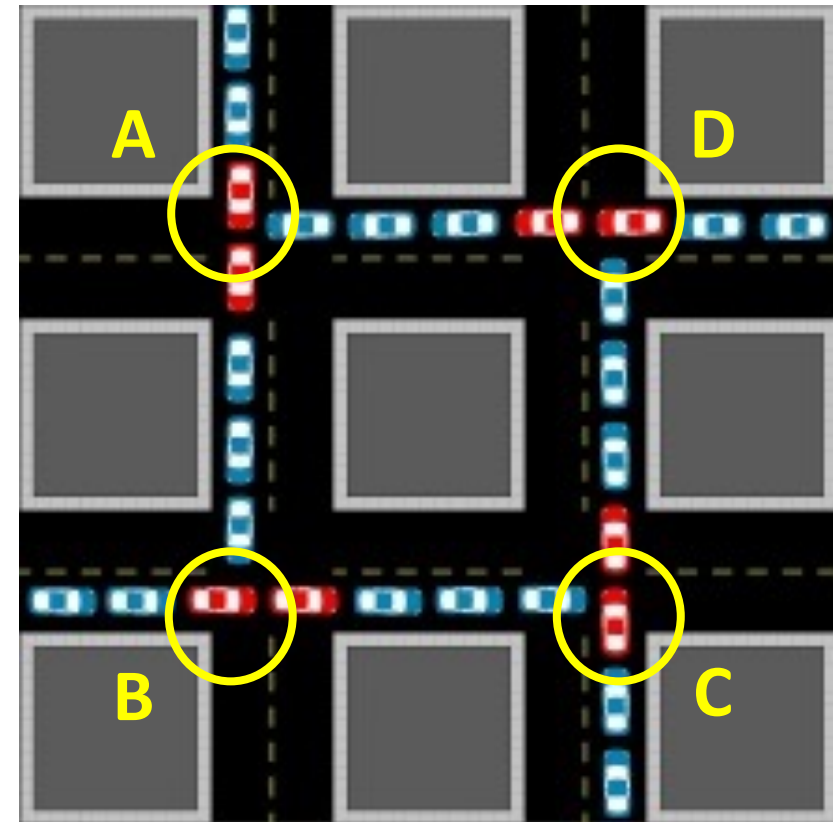
Fix Deadlock With Ordered Resources

In order to fix deadlock, impose an **order** that programs always follow when requesting resources.

For example, maybe Youtube and Zoom must receive the screen lock before they can request the audio. When Youtube gets the screen, it can make a request for the audio while Zoom waits for its turn.

When Youtube is done, it will yield its resources and Zoom will be able to access them.
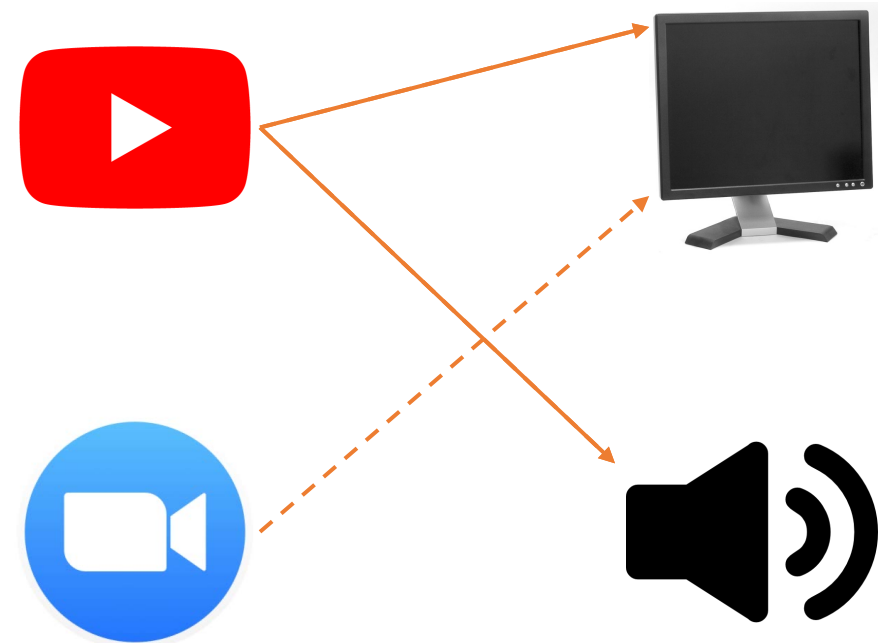
Some Processes Need to Communicate

We can't always guarantee that the processes running concurrently on a computer are independent. If a single program is split into multiple tasks that run concurrently instead, those tasks might need to share partial results as they run. They'll need a way to **communicate** with each other.

Data is shared between processes by **passing messages**. When one task has found a result, it may send it to the other process before continuing its own work.

If one process depends on the result of another, it may need to halt its work while it waits on the message to be delivered. This can slow down the concurrency, as it takes time for data to be sent between cores or computers. **Example:** in tree-summing, a core will need to wait for both calls to finish before it can sum the results.

Generic Parallel Approaches

Writing algorithms that can pass messages is tricky. To make it easier, we use **general algorithmic approaches** that can be adapted for specific tasks.

We'll discuss one common approach today (**pipelining**) and another in the next lecture (**MapReduce**).

Pipelining

Pipelining Definition

One algorithmic process that simplifies parallel algorithm design is **pipelining**. In this process, you start with a task that repeats the same procedure over many different pieces of data.

The steps of the procedure are split across different cores. Each core is like a single worker on an **assembly line**; when it is given a piece of data it executes the step, then passes the result to the next core.

Just like in an assembly line, the cores can run **multiple pieces of data simultaneously** by starting new computations while the others are still in progress.

Demo: Real-Life Pipelining

Let's compare pipelining to sequential work with a real-life race!

We need to generate ten greeting cards. We can divide the process of writing a greeting card into three steps:

1. Write 'Wish you were here!' inside the card
2. Put the card inside an envelope
3. Seal the envelope

What happens if we have one process (person) complete all three tasks vs. having three processes (people) complete the tasks using a pipeline?

Sequential Pizza – 1 worker, 1 oven, 12 steps

Here's an example of pipelining through the lens of line cooking. To make a pizza, we must:

1. Flatten the dough
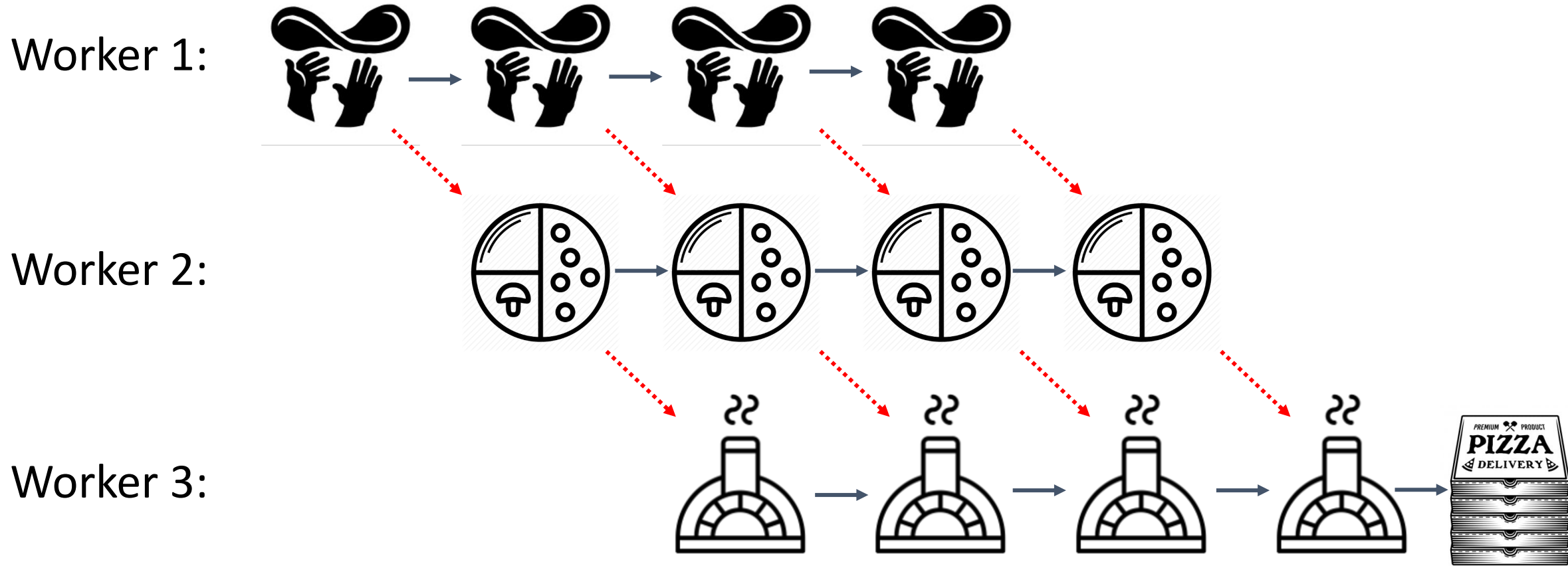2. Apply the toppings
3. Bake in the oven

If we need to make four pizzas without parallelization, it will look like this:
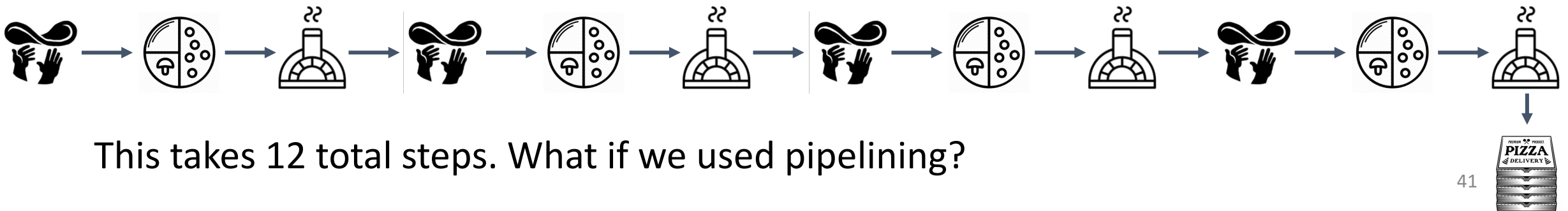
This takes 12 total steps. What if we used pipelining?

Pipelining Pizza - 3 workers, 1 oven, 6 steps

Worker 1:

Worker 2:

Worker 3:

Each worker has **one task**. #1 flattens dough, #2 arranges toppings, #3 bakes in the oven.
There are still 12 total steps, but only **6 time steps** occur.

Rules for Pipelining

When designing a pipeline, it's important to remember that **each step relies on the step that came before it**. You cannot start applying toppings until the dough has been flattened.

Additionally, the length of time that the pipelining process takes **depends on the longest step**. If flattening dough and applying toppings are fast (maybe 5 minutes each) but cooking in the oven is slow (maybe 20 minutes), the whole process will have to wait on the slowest step to conclude.

Benefits of Pipelining

Pipelining is most useful when the number of shared resources is **limited**. For example, you probably use pipelining when doing laundry at home, because you have a limited number of washers and driers to work with!

In computer science, pipelining is used to increase the efficiency of certain operations, like matrix multiplication. It's also used in the Fetch-Decode-Execute cycle, which is how the CPU processes instructions.

Learning Goals

- Recognize and define the following keywords: **concurrency**, **parallel programming**, **CPU**, **scheduler**, **throughput**, **multitasking**, **multiprocessing**, and **deadlock**

- Calculate the **total steps** and **time steps** taken by a parallel algorithm

- Create **pipelines** to increase the efficiency of repeated operations by splitting steps across cores